CHAPTER 7
DESIGN EXAMPLE: BUTTONS AND LIGHTS

In this chapter we'll build a simple I/O device, using what we learned about PC host software in Chapter 6 and what we learned about the responsibilities of an I/O device in Chapters 2 and 3. I chose the simplest design that I could think of so that it would be easier to focus on the method: this design has a single 8-bit input port to which we will connnect 8 buttons and it has a single 8-bit output port to which we will connect 8 LEDs.

Figure 7-1 represents the overall design task. A PC host application program writes blocks of a buffer to an I/O device and reads buffers of data from an I/O device. The format of these data buffers will be known by the application program and by the microcontroller firmware.

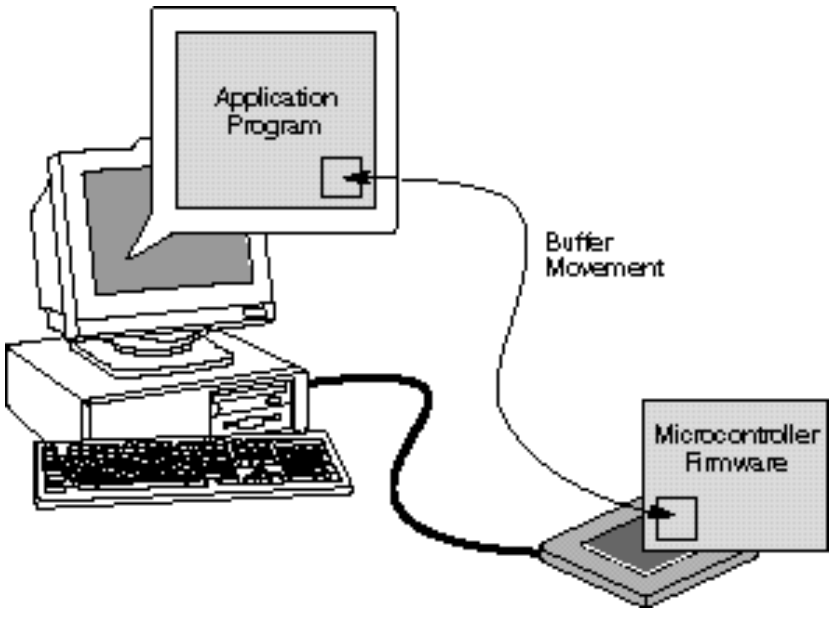


Figure 7-1. Overall task design

We'll eventually have multiple examples, so each I/O device, its hardware, and its matching firmware must be uniquely identified by its Product Code and Name. The PC host application program will first "open" a named I/O device to ensure that the hardware matches the software. The program will also "close" the connection as it exits.

When you develop an I/O device[1], you choose a USB interface, a microcontroller (or a microcontroller with a USB interface), and the required I/O devices (for guidance, see Appendix A). Then you write the subroutines that respond to the commands sent by the PC host.

DESIGN EXAMPLE: BUTTONS AND LIGHTS

This example implements an interface between USB and 8 buttons and 8 lights.

The application goes through various stages of initialization:

- The I/O device is attached and enumerated. The I/O device enumerates as a HID device, and the system creates an entry for it in the HID device list attached to the root Plug and Play node. The second software example in Chapter 6 explained this operation in detail.
- The PC host application starts executing. The application searches through the Plug-and-Play list for "Buttons and Lights." If this is not present, the application prompts the user to attach the I/O device. Because we haven't built our hardware yet, the PC host program is going to wait a long time to detect the presence of our I/O device!

[1] An alternative method using synthesizable cores was discussed in Chapter 4. This chapter will focus on the lower volume implementation using a microcontroller.

Step 1: Design the Hardware

Figure 7-2 shows our first USB I/O device: a simple, bus-powered I/O device consisting of a single 8-bit input port, with "buttons," and a single 8-bit output port, with "lights." I'm using a simple I/O device so we can focus on the **method.**

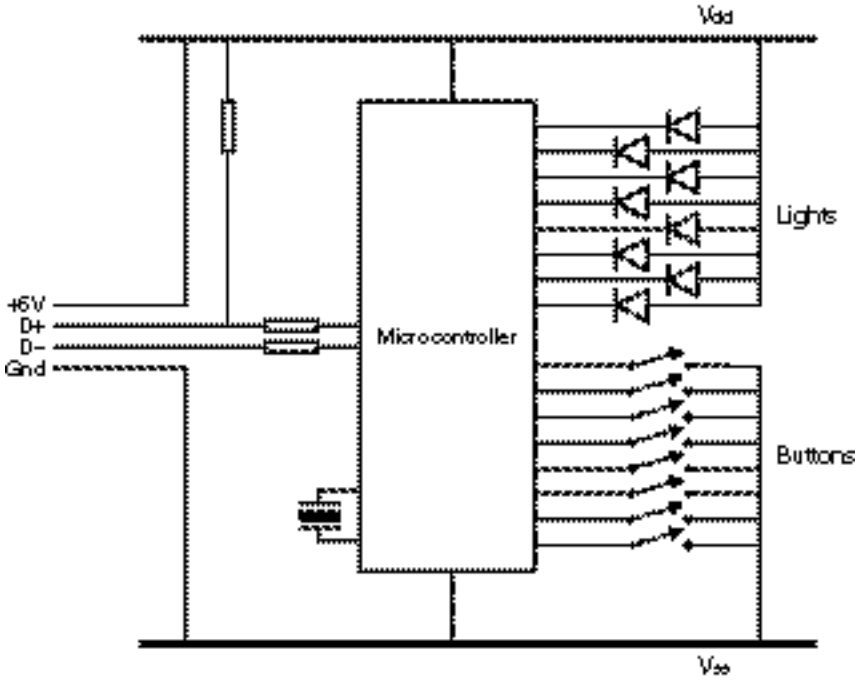


Figure 7-2. Our first USB I/O device

To speed the development along I chose to implement this design example using a pre-built USB development board. I used the USBSIMM board from J. Gordon Electronic Design; this board, shown in Figure 7-3, integrates the Cypress EZ-USB component and other essential circuitry to create a fundamental building block for I/O device designs. A 32K RAM component is included for large data transfers or for very large programs. An 8K EEPROM component allows permanent data, or a program, to be stored. Two 8-bit ports are connected to a Simmbus expansion connector and these will be used for our first example design. A serial port, I2C bus and power are also routed to the Simmbus connector.

Use of the Simmbus enables standard boards to be used in our example designs. Figure 7-3 also shows a DT003 back plane, a DT203 I/O board and a variety of prototyping boards which are available from www.simmstick.com. I only partially populated the DT203 board to include one "buttons" port and one "lights" port. The schematics for these boards are included in the Chapter 7 directory of the CDROM should you want to build your own boards (not recommended for prototyping, get it working first, THEN optimize!)

In this first example, design work consists mostly of writing firmware that is executed during the enumeration phase of initialization. The firmware looks quite verbose, but the good news is that we won't need to change it much for all of the other examples. Let's work through the example slowly so that we fully understand the steps.

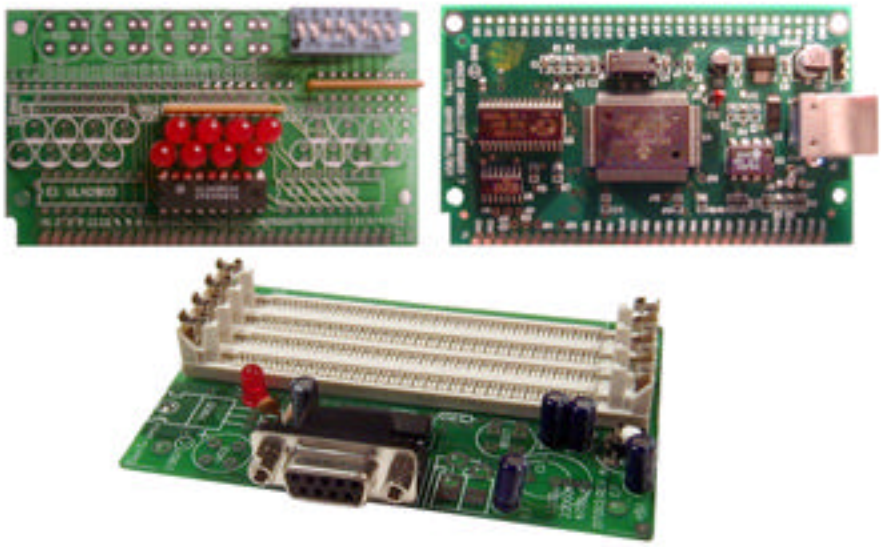


Figure 7-3. Using pre-built modules saves initial development time

Step 2: Complete the Descriptors

Our first step is to complete the descriptors for this HID device. We have one interface and one configuration in this single device. The interface descriptor will point to a HID descriptor that in turn points to our single report. The report descriptor was generated using the HID tool introduced in Chapter 6. Figure 7-4 shows the descriptors for our first example.

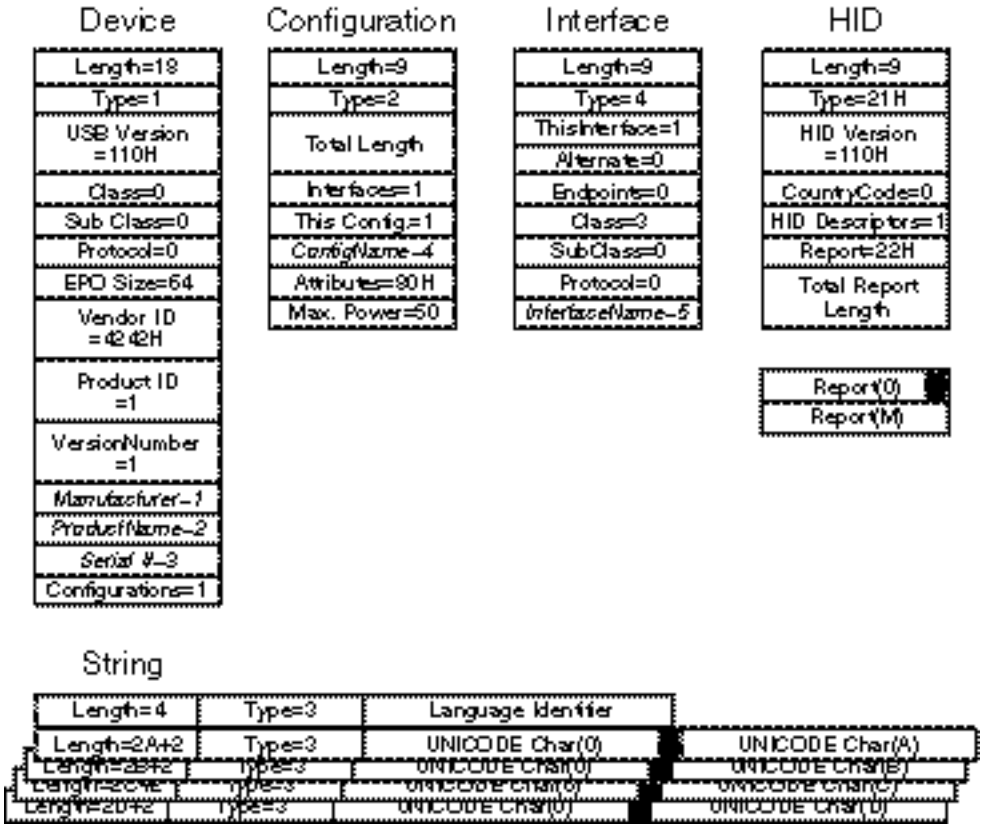


Figure 7-4. Descriptors for our first example

Table 7-1 lists the Standard Request Codes and the Class Request Codes that the PC host may send to our I/O device. Each request can be targeted at the device, an interface, or an endpoint, so the number of possible requests is quite large. Our general response to each request is also listed, but the details will vary. With so few features in this first example, many of the responses will be null subroutines. Our software structure allows for fuller responses in later examples.

Table 7-1. Requests that our I/O device must respond to

Type	Request	Our Action*
Standard	Get_Status	Return current status
Standard	Clear_Feature	Do nothing, we have no clearable features
Standard	Set_Feature	Do nothing, we have no settable features
Standard	Set_Address	Do it
Standard	Get_Descriptor	Return requested descriptor
Standard	Set_Descriptor	Do nothing, our descriptors are static
Standard	Get_Configuration	Return it or 0 if not configured
Standard	Set_Configuration	Do nothing, we have only one
Standard	Get_Interface	Return it
Standard	Set_Interface	Do nothing, we have only one
Standard	Sync_Frame	Report error since we are not an Async device
Standard	all others	Report error via a Stall
Class	Get_Report	Return it
Class	Get_Idle	Return it
Class	Get_Protocol	Do nothing, we are not a boot device
Class	Set_Report	Do it
Class	Set_Idle	Do it
Class	Set_Protocol	Do nothing, we are not a boot device
Class	all others	Report error via a Stall

* "Do Nothing" requires the correct handshaking protocol. If an "Unknown" request is received, we must stall our request queue.

Step 3: Implement Microcontroller Code

Now we're ready to implement the microcontroller code. Any of the microcontrollers presented so far could be used here. Each has a different SIE interface that requires slightly different software, but the overall structure of the solution will be the same.

I have divided the code into five modules (Figure 7-5):

- Two modules define program variables and fixed constants: DECLARE and DTABLE.
- Three modules contain code: MAIN, INTERRUPT, and DECODE.

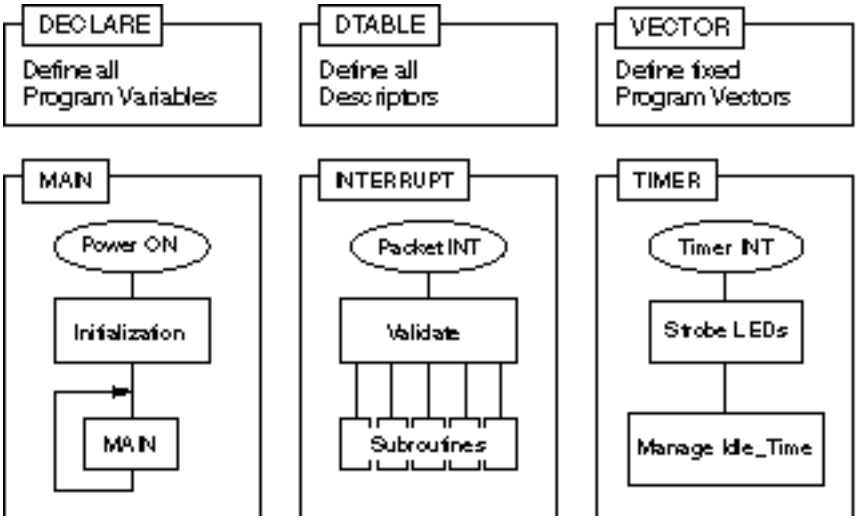


Figure 7-5. Overview of the I/O device firmware

The MAIN module receives control following power-on. After initializing the program variables and microcontroller peripherals, the module executes the MAIN task forever. The MAIN module receives buffers from the PC host, interprets their contents, and acts upon them. This module also generates buffers for the PC host and supplies them on request. Note that the microcontroller never initiates a USB action – it is a slave device that responds to PC host commands.

NOTE

All of the examples in this chapter will be able to use the same INTERRUPT module, because buffer interpretation and creation are handled in MAIN.

The INTERRUPT module does most of the work for the first example, because this module deals with the interrupts signaled by the SIE. It handles all standard requests and most of the HID class requests using the support function DECODE. Two of the HID class requests, SetReport and GetReport, result in data buffers that are passed to the MAIN module for processing.

Figure 7-6 shows how the EZ-USB responds to a control read transfer. The EZ-USB has a particularly smart SIE that validates received packets, generates the correct handshake packet, and interrupts the protocol microcontroller only after a good transaction has been received. The SIE generates separate interrupts for each transaction type and vectors the microcontroller, via a jump table, to the correct interrupt service routine. The EZ-USB also has multiple buffers to simplify the handling of control transfers.

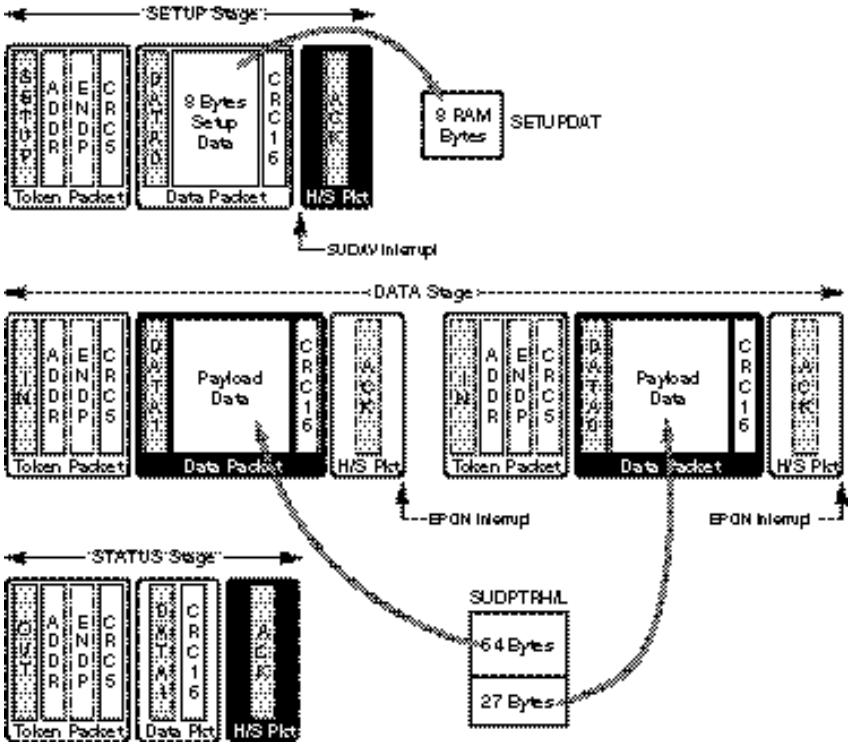


Figure 7-6. Operation of a control read transfer

The handling of all USB command requests is implemented in the DECODE module. This is table driven to allow the easy addition of extra functions required in later examples.

The protocol microcontroller will be interrupted by SIE once a valid control request has been received in the setup buffer. It is the responsibility of the protocol controller to service this request. The request types, detailed in Chapter 2, can be divided into three categories:

- Send data to the PC host.
- Receive data from the PC host.
- Handshake with the PC host.

Figure 7-6 showed the first case of sending data to the PC host. The protocol controller selects the data to send and places its address in a 16-bit SUDPTR register. Writing the low byte of this register prompts the SIE to send the data to the host. If the data is longer than 64 bytes (the maximum size of EP0 on the EZ-USB), then the SIE will send multiple data packets using DATA1 and DATA0 headers alternately. The SIE will also respond with an ACK to the PC host handshake. Because of the richness of the SIE implementation, there is not a lot for the protocol controller to do.

Receiving data from the PC host via an OUT transfer is similarly straightforward (Figure 7-7). The SIE notes the length of the data to be sent by reading bytes 6 and 7 of the SETUP DATA buffer. The SIE then copies received bytes into the OUT0BUF until the required length is reached, generates an EP0OUT interrupt to the protocol controller, and manages a handshake sequence to the PC host.

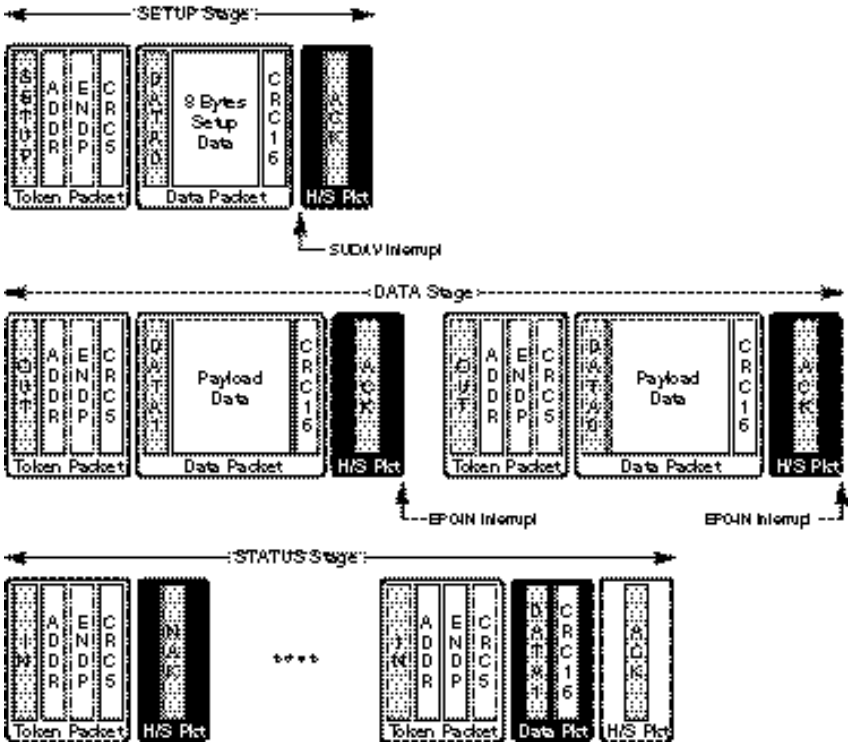
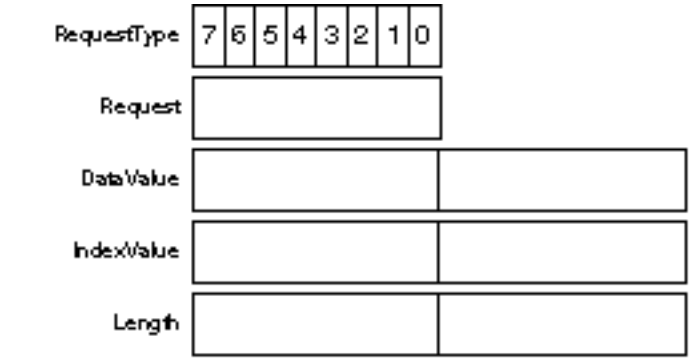


Figure 7-7. Responding to a control write

Thus, to implement an enumeration phase with the SIE interface in this example, the code needs to interpret SETUPDAT after a SUDAV (SetUp_Data_Available) interrupt and respond with data packets or absorb and react to data packets. This example accepts Standard and Class Requests for three recipients: Device, Interface, and Endpoint. The bit field encoding of **RequestType** lets us easily identify requests that are invalid for us. Figure 7-8 (repeated here from Chapter 2 for your convenience) shows that a 1 in bit positions 6, 4, 3, or 2 is not allowed, and our action would be to stall Endpoint 0. Having eliminated the majority of the invalid requests, this example creates an index (Figure 7-9) into the CommandTable using the valid bits from **RequestType** combined with the defined bits of **Request**. Figure 7-10 shows the CommandTable itself.



Definition of Request Type Bit Fields

Bit 7	0 = Transfer Host Data to Device		1 = Transfer Device Data to Host	
Bit 6-5	00 = Standard	01 = Class	10 = Vendor	11 = Reserved
Bit 4-0	0000 = Device	00001 = Interface	00010 = Endpoint	00011 = Other
	All other codes are Reserved			

Figure 7-8. Format of a PC host request

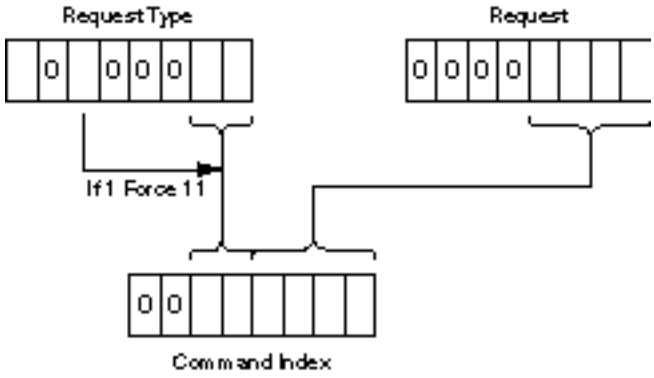


Figure 7-9. Creating an index into the CommandTable

NOTE

The CommandTable in Figure 7-10 includes all possible requests that an HID I/O device needs to respond to, so this approach will apply to all other examples in this chapter.

Requests that are not valid for this particular example will be routed to the INVALID routine. Note that 11 in bits 1:0 of **RequestType** is not valid, so, for coding convenience, the top quarter of the CommandTable is used for class commands.

```
CommandTable:                          ; Table Index
;First 16 commands are for the Device
        Device_Get_Status     ; 00
        Device_Clear_Feature  ; 01
        Invalid                        ; 02
        Device_Set_Feature    ; 03
        Invalid                        ; 04
        Device_Set_Address    ; 05
        Device_Get_Descriptor ; 06
        Device_Set_Descriptor ; 07
        Device_Get_Configuration       ; 08
        Device_Set_Configuration       ; 09
        Invalid                        ; 0A
        Invalid                        ; 0B
        Invalid                        ; 0C
        Invalid                        ; 0D
        Invalid                        ; 0E
        Invalid                        ; 0F
;Next 16 commands are for the Interface
        Interface_Get_Status  ; 10
        Interface_Clear_Feature        ; 11
        Invalid                        ; 12
        Interface_Set_Feature ; 13
        Invalid                        ; 14
        Invalid                        ; 15
        Class_Get_Descriptor  ; 16
        Class_Set_Descriptor  ; 17
        Invalid                        ; 18
        Invalid                        ; 19
        Interface_Get_Interface        ; 1A
        Interface_Set_Interface        ; 1B
        Invalid                        ; 1C
        Invalid                        ; 1D
        Invalid                        ; 1E
        Invalid                        ; 1F
;Next 16 commands are for the Endpoint
        Endpoint_Get_Status   ; 20
        Endpoint_Clear_Feature         ; 21
        Invalid                        ; 22
        Endpoint_Set_Feature  ; 23
        Invalid                        ; 24
        Invalid                        ; 25
        Invalid                        ; 26
        Invalid                        ; 27
        Invalid                        ; 28
        Invalid                        ; 29
        Invalid                        ; 2A
        Invalid                        ; 2B
        Endpoint_Sync_Frame   ; 2C
        Invalid                        ; 2D
        Invalid                        ; 2E
        Invalid                        ; 2F
;Next 16 commands are Class Requests
        Invalid                        ; 30
```

```
                Get_Report          ; 31
                Get_Idle                    ; 32
                Get_Protocol        ; 33
                Invalid                     ; 34
                Invalid                     ; 35
                Invalid                     ; 36
                Invalid                     ; 37
                Invalid                     ; 38
                Set_Report          ; 39
                Set_Idle                    ; 3A
                Set_Protocol        ; 3B
                Invalid                     ; 3C
                Invalid                     ; 3D
                Invalid                     ; 3E
                Invalid                     ; 3F
```

Figure 7-10. CommandTable for requests to HID I/O device

DECODE Module

We must write a subroutine for each command listed in the Command Table. Many of these routines are similar but operate on different data. Rather than describe each routine here, I've added extra comments to the code for the DECODE module (Figure 7-11).

```
; This module is common to all of the examples.
; It decodes the USB Setup Packets and generates appropriate responses.
; Interpretation of Reports is handled by MAIN
;
        CSEG
ServiceSetupPacket:
        MOV     A, RequestType
        MOV     C, ACC.7            ; Bit 7 = 1 means IO device needs to send data
        MOV     SendData, C
        ANL     A, #01011100b       ; IF RequestType[6.4.3.2] = 1 THEN BadRequest
        JNZ     BadRequest
        MOV     A, RequestType      ; IF RequestType[1&0] = 1 THEN BadRequest
        MOV     C, ACC.0
        ANL     C, ACC.1
        JC      BadRequest
        JNB     ACC.5, NotB5        ; IF RequestType[5]=1 THEN CommandIndex[1,0]=[1,1]
        MOV     A, #00000011b
NotB5:  ANL     A, #00000011b       ; Set CommandIndex[5,4] = RequestType[1,0]
        SWAP    A
        MOV     R7, A               ; Save HI nibble of CommandIndex
                                    ; Set CommandIndex[3,0] = Request[3,0]
        MOV     A, Request
        ANL     A, #11110000b       ; Check if Request > 15
        JNZ     BadRequest
        MOV     A, Request
        ANL     A, #00001111b       ; Only 13 are defined today, handle in table
        ORL     A, R7
;                                   ; goto CommandTable(CommandIndex)
CorrectSubroutine:                  ; Jump to the subroutine DPTR is pointing to
        MOV     ReplyCount, #1      ; Set up a default reply
        MOV     ReplyBuffer, #0
        MOV     ReplyBuffer+1, #0
        CLR     SetAddress          ; Clear all flags
        CLR     STALL
        CLR     IsDescriptor
```

```
        MOV     DPTR, #CommandTable
        CALL    BumpDPTR            ; Point to entry
        MOVX    A, @DPTR            ; Get the offset
        MOV     DPTR, #Subroutines
        JMP     @A+DPTR             ; Go to the correct Subroutine

BadRequest:                         ; Decoded a Bad Request, STALL the Endpoint
        SETB    STALL
        RET
                                    ; Support routines
NextDPTR:                           ; Returns (DPTR + byte DPTR is pointing to)
        MOVX    A, @DPTR
BumpDPTR:                           ; Returns (DPTR + ACC)
        ADD     A, DPL
        MOV     DPL, A
        JNC     Skip
        INC     DPH                 ; Need 16 bit arithmetic here
Skip:   RET

; Since the table only contains byte offsets, it is important that all these routines
; are within one page (100H) of Subroutines
; V3.0 - CommandTable moved outside of this one page limited space
CommandTable:
; First 16 commands are for the Device
        DB LOW(Device_Get_Status - Subroutines)
        DB LOW(Device_Clear_Feature - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Device_Set_Feature - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Set_Address - Subroutines)
        DB LOW(Get_Descriptor - Subroutines)
        DB LOW(Set_Descriptor - Subroutines)
        DB LOW(Get_Configuration - Subroutines)
        DB LOW(Set_Configuration - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
; Next 16 commands are for the Interface
        DB LOW(Interface_Get_Status - Subroutines)
        DB LOW(Interface_Clear_Feature - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Interface_Set_Feature - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Get_Class_Descriptor - Subroutines)
        DB LOW(Set_Class_Descriptor - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Get_Interface - Subroutines)
        DB LOW(Set_Interface - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
; Next 16 commands are for the Endpoint
        DB LOW(Endpoint_Get_Status - Subroutines)
        DB LOW(Endpoint_Clear_Feature - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Endpoint_Set_Feature - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
```

```
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Endpoint_Sync_Frame - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
; Next 16 commands are Class Requests
        DB LOW(Invalid - Subroutines)
        DB LOW(Get_Report - Subroutines)
        DB LOW(Get_Idle - Subroutines)
        DB LOW(Get_Protocol - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Set_Report - Subroutines)
        DB LOW(Set_Idle - Subroutines)
        DB LOW(Set_Protocol - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)
        DB LOW(Invalid - Subroutines)

Subroutines:
;
; Many requests are INVALID for this example
Get_Protocol:                       ; We are not a Boot device
Set_Protocol:                       ; We are not a Boot device
Set_Descriptor:                     ; Our Descriptors are static
Set_Class_Descriptor:               ; Our Descriptors are static
Set_Interface:                      ; We only have one Interface
Get_Interface:                      ; We do not have an Alternate setting
Set_Idle:                           ; V3.0 Optional command, not supported
Get_Idle:                           ; V3.0 Optional command, not supported
Device_Set_Feature:                 ; We have no features that can be set or cleared
Interface_Set_Feature:              ; We have no features that can be set or cleared
Endpoint_Set_Feature:               ; We have no features that can be set or cleared
Endpoint_Clear_Feature:             ; V3.0 We have no features that can be set or cleared
Device_Clear_Feature:               ; We have no features that can be set or cleared
Interface_Clear_Feature:            ; We have no features that can be set or cleared
Endpoint_Sync_Frame:                ; We are not an Isonchronous device
Invalid:                            ; Invalid Request made, STALL the Endpoint
        SETB    STALL
Reply:  RET
Set_Address:                        ; Set the address that the SIE will respond to
        SETB    SetAddress
        RET
Set_Report:                         ; Host wants to sent us a Report.
; The ONLY case in this example where host sends data to us
        JNB     Configured, Invalid ; Need to be Configured to do this command
        CALL    GetOutputReport     ; Handled in EZUSB.A51
        JMP     ProcessOutputReport ; RETurn via this subroutine
Get_Report:                         ; Host wants a Report
        JNB     Configured, Invalid ; Need to be Configured to do this command
        MOV     ReplyBuffer, #42H   ; Reply with a recognizable (arbitary) value
        RET
Get_Configuration:                  ; Respond with CurrentConfiguration
        MOV     ReplyBuffer, CurrentConfiguration
        RET
Device_Get_Status:                  ; Only two bits of Device Status are defined
        MOV     ReplyBuffer, #1     ; Bit 1=Remote Wakeup(=0),
                                    ; Bit 0=Self Powered(=1)
        RET
Interface_Get_Status:               ; Interface Status is currently defined as 0
Endpoint_Get_Status:
```

```
        MOV     ReplyCount, #2          ; Need a two byte 0 response
        RET
Set_Configuration:                      ; Valid values are 0 and 1
        MOV     A, wValueLow
        JZ      Deconfigured
        DEC     A
        JNZ     Invalid
        SETB    Configured
        MOV     CurrentConfiguration, #1
        RET
Deconfigured:
        CLR     Configured
        MOV     CurrentConfiguration, A
        RET
Get_Descriptor:                         ; Host wants to know who/what we are
        SETB    IsDescriptor
        MOV     A, wValueHigh
        DEC     A                       ; Valid Values are 1, 2 and 3
        MOV     DPTR, #DeviceDescriptor
        JZ      ReturnLength
        DEC     A
        MOV     DPTR, #ConfigurationDescriptor
        JNZ     TryString
        MOV     A, #ConfigLength
        RET
TryString:
        DEC     A
        JNZ     Invalid
; Request is for a String Descriptor
        MOV     DPTR, #String0          ; Point to String 0
        MOV     A, wValueLow            ; Get String Index
NextString:
        JZ      ReturnLength
        MOV     R7, A                   ; Save String Index
        CALL    NextDPTR
        MOVX    A, @DPTR                ; Get the String Length (= 0 means we're at
Backstop)
        JZ      Invalid                 ; Asked for a string I don't have
        MOV     A, R7
        DEC     A
        JMP     NextString              ; Check if we are there yet
Get_Class_Descriptor:                   ; Valid values are 21H, 22H, 23H
        SETB    IsDescriptor
        MOV     A, wValueHigh
        CLR     C
        SUBB    A, #21H
        MOV     DPTR, #HIDDescriptor
        JZ      ReturnLength
        DEC     A
        MOV     DPTR, #ReportDescriptor
        JZ      ReturnRDlength
;       DEC     A                       ; This example has no Physical Descriptors
;       JZ      Send_Physical_Descriptor
        JMP     Invalid
;
ReturnLength:
        MOVX    A, @DPTR                ; Get Descriptor Length (first byte)
        RET
ReturnRDlength:                         ; Report Descriptor is different format
        MOV     A, #ReportLength
        RET
; Error check: this MUST be on within a page of Subroutines
WithinSamePage EQU $ - Subroutines
```

Figure 7-11. DECODE module

MAIN Module

Figure 7-12 shows the MAIN module. Most of the module consists of system and variable initialization. This initialization code will be reusable in many of the other examples. The MAIN module also includes the service routines for processing the Input Report and creating the Output Report—these routines are the ones that will change with each of the examples.

```
; This module initializes the microcontroller then executes MAIN forever
; It is hardware dependant

Reset:   MOV     SP, #0DFH             ; Initialize the Stack
         MOV     PageReg, #7FH         ; Allows MOVX Ri to access EZ-USB memory

         MOV     R0, #Low(USBControl)  ; Simulate a disconnect
         MOVX    A, @R0
         ANL     A, #11110011b         ; Clear DISCON, DISCOE
         MOVX    @R0, A
         CALL    Wait200msec           ; Give the host time to react
         MOVX    A, @R0                ; Reconnect with this new identity
         ORL     A, #00000110b         ; Set DISCOE to enable pullup resistor
         MOVX    @R0, A                ; Set RENUM so that 8051 handles USB requests
         CLR     A
         MOV     FLAGS, A              ; Start in Default state
TurnOffLEDs:
         MOV     LEDValue, A
         MOV     Old_Buttons, A
         INC     A                     ; = 1
         MOV     LEDstrobe, A
Initialize4msecCounter:
         MOV     Msec_counter, A
InitializeIOSystem:                    ; Setup for Simmbus A=input, B=output
                                       ; C=External RD#,WR#,TD0,TR0
         MOV     R0, #LOW(PortA_Config); PageReg = 7F = HIGH(PortA_Config)
         MOV     R1, #LOW(PortA_OE)
         CLR     A
         MOVX    @R0, A                ; No alternate functions
         MOVX    @R1, A                ; Enable PortA for Input
         INC     R0                    ; Point to PortB_Config
         INC     R1                    ; Point to PortB_OE
         MOVX    @R0, A                ; No alternate functions
         CPL     A                     ; = 0FFH
         MOVX    @R1, A                ; Enable PortB for Output
         INC     R0                    ; Point to PortC_Config
         INC     R1                    ; Point to PortC_OE
         MOV     A, #11000011b
         MOVX    @R0, A                ; Alternate functions on [7,6,1,0]
         MOV     A, #11000010b
         MOVX    @R1, A                ; Most alternate functions are outputs
InitializeInterruptSystem:             ; First initialize the USB level
         MOV     A, #00000001b
         MOV     R0, #LOW(IN07IEN)
         MOVX    @R0, A                ; Enable interrupts from EP0IN only
         INC     R0
         CLR     A
         MOVX    @R0, A                ; Disable interrupts from OUT Endpoints 0-7
         INC     R0
         MOV     A, #00010011b         ; Enable USBReset, SOF and SUDAV INTs
         MOVX    @R0, A
         INC     R0
         MOV     A, #00000001b
         MOVX    @R0, A                ; Enable Auto Vectoring for USB interrupts
                                       ; Now enable the main level
         MOV     EIE, #00000001b       ; Enable INT2 = USB Interrupt (only)
```

```
        MOV     EI, #10010000b        ; Enable interrupt subsystem
                                      ; (and Ser0 for dScope)

; Initialization Complete.
;
MAIN:   NOP                           ; Not much of a main loop for this example
        JMP     MAIN                  ; All actions are initiated by interrupts
; We are a slave, we wait to be told what to do

Wait200msec:
        MOV     R7, #200
Wait1msec:                            ; A delay loop
        MOV     DPS, #0               ; Select primary DPTR
        MOV     DPTR, #-1200
More:   INC     DPTR                  ; 3 cycles
        MOV     A, DPL                ; + 2
        ORL     A, DPH                ; + 2
        JNZ     More                  ; + 3 = 10 cycles x 1200 = 1msec
        DJNZ    R7, Wait1msec
        RET

ProcessOutputReport:                  ; A Report has just been received
; The report is only one byte long in this first example
; It contains a new value for the LEDs
        MOV     DPTR, #EP0OutBuffer   ; Point to the Report
        MOVX    A, @DPTR              ; Get the Data
        MOV     LEDValue, A           ; Update the local variable
        RET

CreateInputReport:                    ; Called from TIMER which detected the need
; The report is only one byte long in this first example
; It contains a new value for the Buttons
        JNB     Configured, NoReport  ; Must be Configured to create reports
        MOV     DPTR, #EP1InBuffer    ; Point to the buffer
        MOVX    @DPTR, A              ; Update the Report
        MOV     DPTR, #IN1ByteCount
        MOV     A, #1
        MOVX    @DPTR, A              ; Endpoint 1 now 'armed', next IN will get data
NoReport:
        RET
```

Figure 7-12. MAIN module

DECLARE Module

The DECLARE module is straightforward to code (Figure 7-13).

```
; This module declares the variables and constants used in the examples
; It is common to all of the examples
;
; Declare Special Function Registers used
EI              DATA 0A8H
EIE             DATA 0E8H             ; EZ-USB specific
EXIF            DATA 091H             ; EZ-USB specific
EICON           DATA 0D8H             ; EZ-USB specific
PageReg         DATA 092H             ; EZ-USB specific, used with MOVX @Ri
DPS             DATA 086H             ; EZ-USB specific, used with dual data pointers
;
; "External" memory locations used, EZ-USB specific
; Note that most of these variables are in Page 7FH
SETUPDAT        EQU 07FE8H
SUDPTR          EQU 07FD4H
```

```
EP0Control       EQU 07FB4H
EP0InBuffer      EQU 07F00H
EP0OutBuffer     EQU 07EC0H           ; Not in Page 7FH
EP1InBuffer      EQU 07E80H           ; Not in Page 7FH
IN0ByteCount     EQU 07FB5H
Out0ByteCount    EQU 07FC5H
IN1ByteCount     EQU 07FB7H
IN07IEN          EQU 07FACH
IN07IRQ          EQU 07FA9H
OUT07IEN         EQU 07FADH
OUT07IRQ         EQU 07FAAH
USBIEN           EQU 07FAEH
USBIRQ           EQU 07FABH
USBControl       EQU 07FD6H
I2CData          EQU 07FA6H
I2CControl       EQU 07FA5H
PortA_Config     EQU 07F93H
PortB_Config     EQU 07F94H
PortC_Config     EQU 07F95H
PortA_OUT        EQU 07F96H
PortB_OUT        EQU 07F97H
PortC_OUT        EQU 07F98H
PortA_PINS       EQU 07F99H
PortB_PINS       EQU 07F9AH
PortC_PINS       EQU 07F9BH
PortA_OE         EQU 07F9CH
PortB_OE         EQU 07F9DH
PortC_OE         EQU 07F9EH
;
; Byte Variables

        DSEG     AT 20H
FLAGS:           DS 1                 ; This register is bit-addressable
; Bit Variables
Configured       EQU FLAGS.0          ; Is this device configured
STALL            EQU FLAGS.1          ; Need to STALL endpoint 0
SendData         EQU FLAGS.2          ; Need to send data to PC Host
IsDescriptor     EQU FLAGS.3          ; Enable a shortcut reply
SetAddress       EQU FLAGS.4          ; Set the SIE address
;
MonitorSpace:    DS 1FH               ; Used by Dscope
DataSpace:
ReplyCount:      DS 1                 ; Byte count for following buffer
ReplyBuffer:     DS 2                 ; Buffer for immediate reply
CurrentConfiguration:
                 DS 1                 ; Some examples support > 1 configurations
SaveDPH:         DS 1                 ; Needed to save Descriptor Pointer ..
SaveDPL:         DS 1                 ; .. for descriptors > EP0Size
SaveLength:      DS 1                 ; Number of bytes still to send
SetupData:                            ; Buffer in direct access memory
RequestType:     DS 1
Request:         DS 1
wValueLow:       DS 1
wValueHigh:      DS 1
wIndexLow:       DS 1
wIndexHigh:      DS 1
wLengthLow:      DS 1
wLengthHigh:     DS 1
;
Old_Buttons:     DS 1                 ; Used by BAL: stores current button position
LEDstrobe:       DS 1                 ; Used by BAL: strobe one LED on at a time
LEDvalue:        DS 1                 ; Used by BAL: stores current LED value
Msec_Counter:    DS 1                 ; Used by BAL: counts up to 4 msec
;
```

Figure 7-13. DECLARE module

DTABLE Module

The DTABLE module is straightforward to code (Figure 7-14).

```
; This module declares the descriptors
;
; This example has one Device Descriptor with:
;       One Configuration - single IN port and single OUT port
;       One Interface - there is only one method of accessing the ports
;       One HID Descriptor - to make PC host software simpler
;       One Endpoint Descriptor - for HID Input Reports
;       One Report Descriptor - one byte IN and one byte OUT reports
;       Multiple Sting Descriptors - to aid the user
;
        CSEG
DeviceDescriptor:
        DB      18, 1                   ; Length, Type
        DB      10H, 1                  ; USB Rev 1.1 (=0110H, low=10H, High=01H)
        DB      0, 0, 0                 ; Class, Subclass and Protocol
        DB      EP0Size
        DB      42H, 42H, 1, 42H, 0, 1; Vendor ID, Product ID and Version
        DB      1, 2, 0                 ; Manufacturer, Product & Serial# Names
        DB      1                       ; #Configs
ConfigurationDescriptor:
        DB      9, 2                    ; Length, Type
        DB      LOW(ConfigLength), HIGH(ConfigLength)
        DB      1, 1, 0                 ; #Interfaces, Configuration#, Config. Name
        DB      10000000b               ; Attributes = Bus Powered
        DB      250                     ; Max. Power is 250x2 = 500mA
InterfaceDescriptor:
        DB      9, 4                    ; Length, Type
        DB      0, 0, 1                 ; No alternate setting, HID uses EP1
        DB      3                       ; Class = Human Interface Device
        DB      0, 0                    ; Subclass and Protocol
        DB      0                       ; Interface Name
HIDDescriptor:
        DB      9, 21H                  ; Length, Type
        DB      0, 1                    ; HID Class Specification compliance
        DB      0                       ; Country localization (=none)
        DB      1                       ; Number of descriptors to follow
        DB      22H                     ; And it's a Report descriptor
        DB      LOW(ReportLength), HIGH(ReportLength)
EndpointDescriptor:
        DB      7, 5                    ; Length, Type
        DB      10000001b               ; Address = IN 1
        DB      00000011b               ; Interrupt
        DB      EP0Size, 0              ; Maximum packet size
        DB      100                     ; Poll every 0.1 seconds
ConfigLength    EQU $ - ConfigurationDescriptor

ReportDescriptor:                       ; Generated with HID Tool, copied to here
        DB      6, 0, 0FFH              ; Usage_Page (Vendor Defined)
        DB      9, 1                    ; Usage (I/O Device)
        DB      0A1H, 1                 ; Collection (Application)
        DB      19H, 1                  ;   Usage_Minimum (Button 1)
        DB      29H, 8                  ;   Usage_Maximum (Button 8)
        DB      15H, 0                  ;   Logical_Minimum (0)
        DB      25H, 1                  ;   Logical_Maximum (1)
        DB      75H, 1                  ;   Report_Size (1)
        DB      95H, 8                  ;   Report_Count (8)
        DB      81H, 2                  ;   Input (Data,Var,Abs)
        DB      19H, 1                  ;   Usage_Minimum (Led 1)
        DB      29H, 8                  ;   Usage_Maximum (Led 8)
        DB      91H, 2                  ;   Output (Data,Var,Abs)
        DB      0C0H                    ; End_Collection
```

```
ReportLength     EQU $-ReportDescriptor

String0:                              ; Declare the UNICODE strings
        DB       4, 3, 9, 4           ; Only English language strings supported
String1:                              ; Manufacturer
        DB       (String2-String1),3  ; Length, Type
        DB       "U",0,"S",0,"B",0," ",0,"D",0,"e",0,"s",0,"i",0,"g",0,"n",0," ", 0
        DB       "B",0,"y",0," ",0,"E",0,"x",0,"a",0,"m",0,"p",0,"l",0,"e",0
String2:                              ; Product Name
        DB       (EndOfDescriptors-String2),3
        DB       "B",0,"u",0,"t",0,"t",0,"o",0,"n",0,"s",0," ",0
        DB       "&",0," ",0,"L",0,"i",0,"g",0,"h",0,"t",0,"s",0
EndOfDescriptors:
        DB       0                    ; Backstop for String Descriptors
```

Figure 7-14. DTABLE module

INTERRUPT Module

The INTERRUPT module is straightforward to code (Figure 7-15).

```
; This module contains all the EZUSB-specific hardware code
; This module also contains all of the interrupt vector declarations and
; the first level interrupt servicing (register save, call subroutine,
; clear interrupt source, restore registers, return)
; Suspend and Resume are handled totally in this module
;
; A Reset sends us to Program space location 0
        CSEG AT  0                    ; Code space
        USING 0                       ; Reset forces Register Bank 0
        LJMP     Reset
;
; The interrupt vector table is also located here
; EZ-USB has two levels of USB interrupts:
; 1-the main level is described in this table (at ORG 43H)
; 2-there are 21 sources of USB interrupts and these are described in USB_ISR
; This means that two levels of acknowledgement and clearing will be required
        ORG      43H
        LJMP     USB_ISR              ; Auto Vector will replace byte 45H

        ORG      1200H                ; Load above monSIO0.hex
USB_ISR:LJMP     SUDAV_ISR
        DB       0                    ; Pad entries to 4 bytes
        LJMP     SOF_ISR
        DB       0
        LJMP     SUTOK_ISR
        DB       0
        LJMP     Suspend_ISR
        DB       0
        LJMP     USBReset_ISR
        DB       0
        LJMP     Reserved
        DB       0
        LJMP     EP0In_ISR
; End of Interrupt Vector tables

; When a feature is used insert the required interrupt processing here
; The example use only used Endpoints 0 and 1 and also SOF for timing
Reserved:
SUTOK_ISR:
Suspend_ISR:
WakeUp_ISR:                           ; Not using external WAKEUP in these examples
EP0Out_ISR:
```

```
Not_Used:                           ; Should not get any of these
        RETI

ClearINT2:                          ; Tell the hardware that we're done
        MOV     A, EXIF
        CLR     ACC.4               ; Clear the Interrupt 2 bit
        MOV     EXIF, A
        RET

USBReset_ISR:                       ; Bus has been Reset, move to DEFAULT state
        CLR     Configured
        MOV     DPTR, #(1000H OR LOW(USBIRQ))
ExitISR:                            ; Common exit for all ISR's
; On entry DPH = Interrupt ID, DPL = LOW(Interrupt Register)
        CALL    ClearINT2
        MOV     A, #7FH             ; EZ-USB I/O Register Page
        XCH     A, DPH
        MOVX    @DPTR, A            ; Clear source of interrupt
        RETI


EP0In_ISR:                          ; A prepared packet has been read by PC host
        MOV     A, SaveLength       ; Do I have any more data to send?
        JZ      NoMoreToSend
        MOV     DPH, SaveDPH        ; Retreive descriptor pointer
        MOV     DPL, SaveDPL
        CALL    SendNextPieceOfDescriptor
NoMoreToSend:
        MOV     DPTR, #(100H OR LOW(IN07IRQ))
        JMP     ExitISR

SOF_ISR:                            ; A Start-Of-Frame packet has been received
; This routine services the real time interrupt
; It is also responsible for the "real world" buttons and lights
;
ServiceTimerRoutine:
        DJNZ    Msec_counter, Done  ; Only need to check every 4msec
        MOV     Msec_counter, #4    ; Reinitialize
; LED task
        MOV     A, LEDValue
        MOV     DPTR, #PortB_Out
        MOVX    @DPTR, A            ; Update the real world
;
; Create an Input Report from the Buttons value
; This will be continually overwritten while the PCHost is not polling for data
ReadButtons:
        MOV     DPTR, #PortA_Pins
        MOVX    A, @DPTR
        CALL    CreateInputReport
Done:   MOV     DPTR, #(200H OR LOW(USBIRQ))
        JMP     ExitISR

SUDAV_ISR:                          ; A Setup packet has been received
        MOV     SaveLength, #0      ; Clear any pending transactions (if any)
        MOV     DPTR, #SETUPDAT     ; Copy packet to direct access memory
        MOV     R0, #SetupData
        MOV     R7, #8
CopySD: MOVX    A, @DPTR
        MOV     @R0, A
        INC     DPTR
        INC     R0
        DJNZ    R7, CopySD
        CALL    ServiceSetupPacket  ; Handle the decode of the Setup packet
; if SetAddress { Update SIE address }  // NOP on EZ-USB
; if STALL { Stall the endpoint }
; if SendData {
;       if IsDescriptor { send DPTR->descriptor, A = length }
```

```
;       else { send ReplyBuffer }
;       }
        JB      STALL, SendSTALL
        JNB     SendData, HandShake
        JB      IsDescriptor, LoadEP0
                                        ; Send data in ReplyBuffer
        MOV     DPTR, #EP0InBuffer+1
        MOV     R0, #ReplyBuffer+1
        MOV     R7, #2                  ; Copy the two byte buffer
CopyRB: MOV     A, @R0
        MOVX    @DPTR, A
        DEC     DPL
        DEC     R0
        DJNZ    R7, CopyRB
        MOV     A, @R0                  ; Get BufferCount
SendEP0InBuffer:
        MOV     DPTR, #In0ByteCount
StartXfer:
        MOVX    @DPTR, A                ; This write initiates the transfer
HandShake:                              ; Handshake with host
        MOV     R7, #00000010b          ; Set HSNAK to tell the SIE that we're done
SetEP0Control:
        MOV     DPTR, #EP0Control
        MOVX    A, @DPTR
        ORL     A, R7
        MOVX    @DPTR, A                ; We're done
        MOV     DPTR, #(100H OR LOW(USBIRQ))
        JMP     ExitISR
SendSTALL:                              ; Invalid Request was received
        MOV     R7, #00000011b          ; Set EP0STALL and HSNAK
        JMP     SetEP0Control
LoadEP0:                                ; Send the data pointed to by DPTR
        MOV     R7, A                   ; Save LENGTH
; Need to return the smaller of "Requested Length" and "Actual Length"
; If "Requested Length" > 255 then use "Actual Length"
; There are no descriptors > 255 in this example
        MOV     A, wLengthHigh
        JNZ     UseActual
        MOV     A, R7                   ; Retreive LENGTH
        CLR     C
        SUBB    A, wLengthLow
        MOV     A, wLengthLow           ; This does not affect Carry
        JNC     UsewLengthLow
UseActual:
        MOV     A, R7
UsewLengthLow:
SendNextPieceOfDescriptor:              ; DPTR -> Descriptor to be sent
        MOV     R7, A                   ; Save LENGTH again
        MOV     SaveLength, #0          ; Default case, overwrite if necessary
; Do I have more than a single packet to send?
        CLR     C
        SUBB    A, #EP0Size
        JC      SendPacket
; Need to send multiple packets.
; Calculate and save address of next packet, send next packet now
        MOV     SaveLength, A           ; Send these next time
        MOV     R7, #EP0Size
        PUSH    DPH                     ; Save current pointer
        PUSH    DPL
        MOV     A, R7                   ; Retreive length
        CALL    BumpDPTR
        MOV     SaveDPH, DPH
        MOV     SaveDPL, DPL
        POP     DPL
        POP     DPH
SendPacket:
        MOV     A, R7                   ; Retreive length
```

```
        MOV     R6, A                   ; Save length in R6 for move
        MOV     R0, #LOW(EP0InBuffer) ; PageReg = 7FH = HIGH(EP0InBuffer)
CopySTD:MOVX    A, @DPTR
        MOVX    @R0, A
        INC     DPTR
        INC     R0
        DJNZ    R6, CopySTD
        MOV     A, R7                   ; Retrieve LENGTH
        JMP     SendEP0InBuffer

GetOutputReport:                        ; Wait for this, it's next on USB
        MOV     DPTR, #Out0ByteCount  ; Enable EP0OutBuffer to receive data
        MOVX    @DPTR, A              ; Any value will do
       MOV      DPTR, #EP0Control     ; Wait for valid data in EP0OutBuffer
Wait40: MOVX    A, @DPTR
        ANL     A, #00001000b         ; Check OUTBSY
        JNZ     Wait40
        RET
```

Figure 7-15. INTERRUPT module

Included on the companion CDROM, in the TOOLS directory, is a set of 8051 Development Tools from Keil Corporation. These tools include evaluation 8051 assembler, linker, locator and a C compiler and their dScope debugger. They are fully functional except that they can only generate programs up to 2K Bytes maximum. Our example uses just under 1K so these tools are fine. When you start developing larger programs then you will need to purchase the full toolset from Keil. These Keil tools run on a Windows PC and create 8051 executable code so they are called cross-development tools. You are using the PC to generate software that will run on another system - the USBSIMM in this case.

Open the BAL project file with the Keil Project Manager. BAL.A51 is a header file that $INCLUDES the other source files. I partitioned the software into different modules so that it will be easier for you to modify and extend the design for your own applications.

You may want to change the way that input reports are created or how output reports interact with the real world. In an effort to provide code which can be easily modified, these routines have all been placed in a single module (EZMAIN.A51).

When you BUILD the project, a BAL.LST file will be generated for you to read and a BAL.HEX file will be generated that must be loaded into the USBSIMM board.

Step 4: Attaching an Empty USBSIMM to the PC Host

We have a chicken-and-egg situation with the EZ-USB component. Its program memory is RAM-based and must be downloaded before the 8051 microcontroller can start execution. The EZ-USB will enumerate using the default Cypress Vendor ID and Product ID (0547, 2131 respectively). It is our responsibility to recognize that this empty device is being attached to the PC and we should download something useful into the program memory. Refer to the EZ-USB Technical Reference Manual Chapter 5 (downloadable from www.cypress.com) for more details on this renumeration process. During enumeration the PC host uses the VID and PID provided by an I/O device to identify which device driver should be loaded. The INF file shown below specifies the LoadEZ.sys autoloader which will subsequently load 05472131.hex onto the empty USBSIMM board (refer to the LoadEZ documentation for further details).

```
[Version]
signature="$CHICAGO$"
Class=UNKNOWN
Provider=%USBDbyE%
LayoutFile=LAYOUT.INF

[Manufacturer]
%USBDbyE%=USBDbyE

[PreCopySection]
HKR,,NoSetupUI,,1

[DestinationDirs]
DefaultDestDir=10,system32\drivers

[USBDbyE]
; This is the VID/PID for an empty EZ-USB development board.
; The VID is 0547H = Anchorchips (now Cypress Semiconductor) The PID is 2131
%USB\VID_0547&PID_2131.DeviceDesc%=EZUSBLR, USB\VID_0547&PID_2131

[EZUSBLR]
CopyFiles=EZUSBLR.Copy
AddReg=EZUSBLR.AddReg

[EZUSBLR.Copy]
LoadEZ.sys

[EZUSBLR.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,LoadEZ.sys

[Strings]
USBDbyE="USB Design By Example"
USB\VID_0547&PID_2131.DeviceDesc="Empty EZ-USB Development Board"
```

Figure 7-16 Empty.INF file that will match the empty USBSIMM board

The 05472131.hex file provided on the CDROM is a renamed monSIO0.hex from the Keil tool suite (LoadEZ.sys requires the filename to be vvvvpppp.hex where vvvv=Vendor ID and pppp = Product ID, as presented in Chapter 6). So the program loaded onto an empty USBSIMM board is the Keil monitor program called dScope.

Step 5: Debug IO device hardware and software

Once connected to the PC host an empty USBSIMM board will download the Keil dScope monitor and will start executing it. The monitor waits for commands to be sent to it via Serial Port 0.
It is now time to connect a straight-through cable from a PC host COM port and Serial Port 0 on the USBSIMM + DT003. Start the PC host side of the Keil dScope debugger and change the default connection to mon51.dll as shown below. Communication at 19200 will be made via the RS232 link to the USBSIMM and the now downloaded monitor will sign-on.

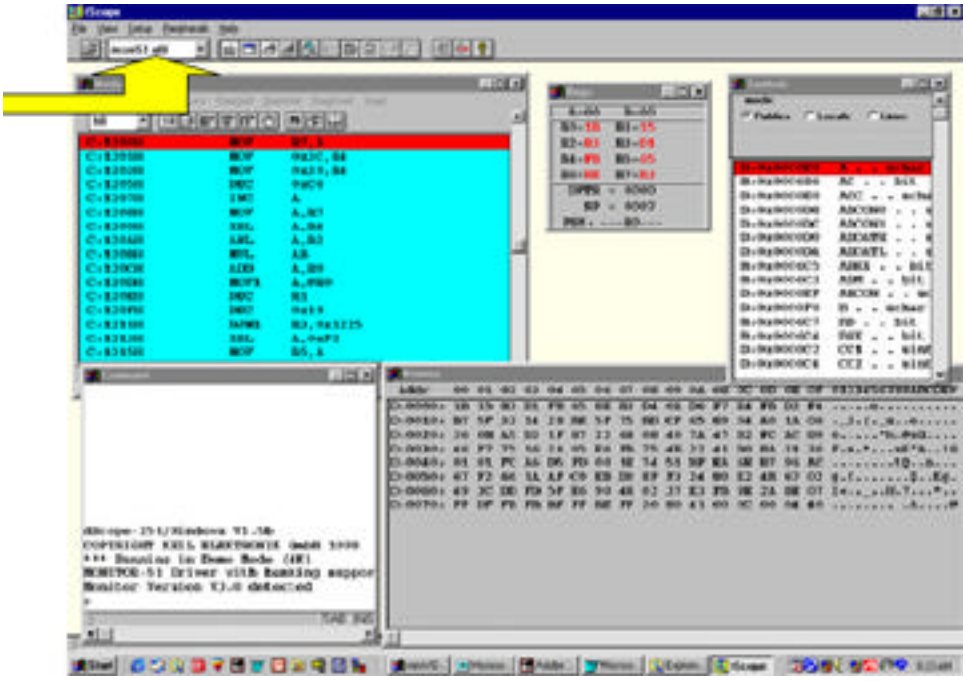


Figure 7-17 Setting dScope to communicate via the serial port

The dScope monitor has a variety of commands such as display memory and registers and the ability to single step or run programs with breakpoints. It will also allow you to download other programs to test. More information is contained within the User Manual that was downloaded with the Development Tool Suite.

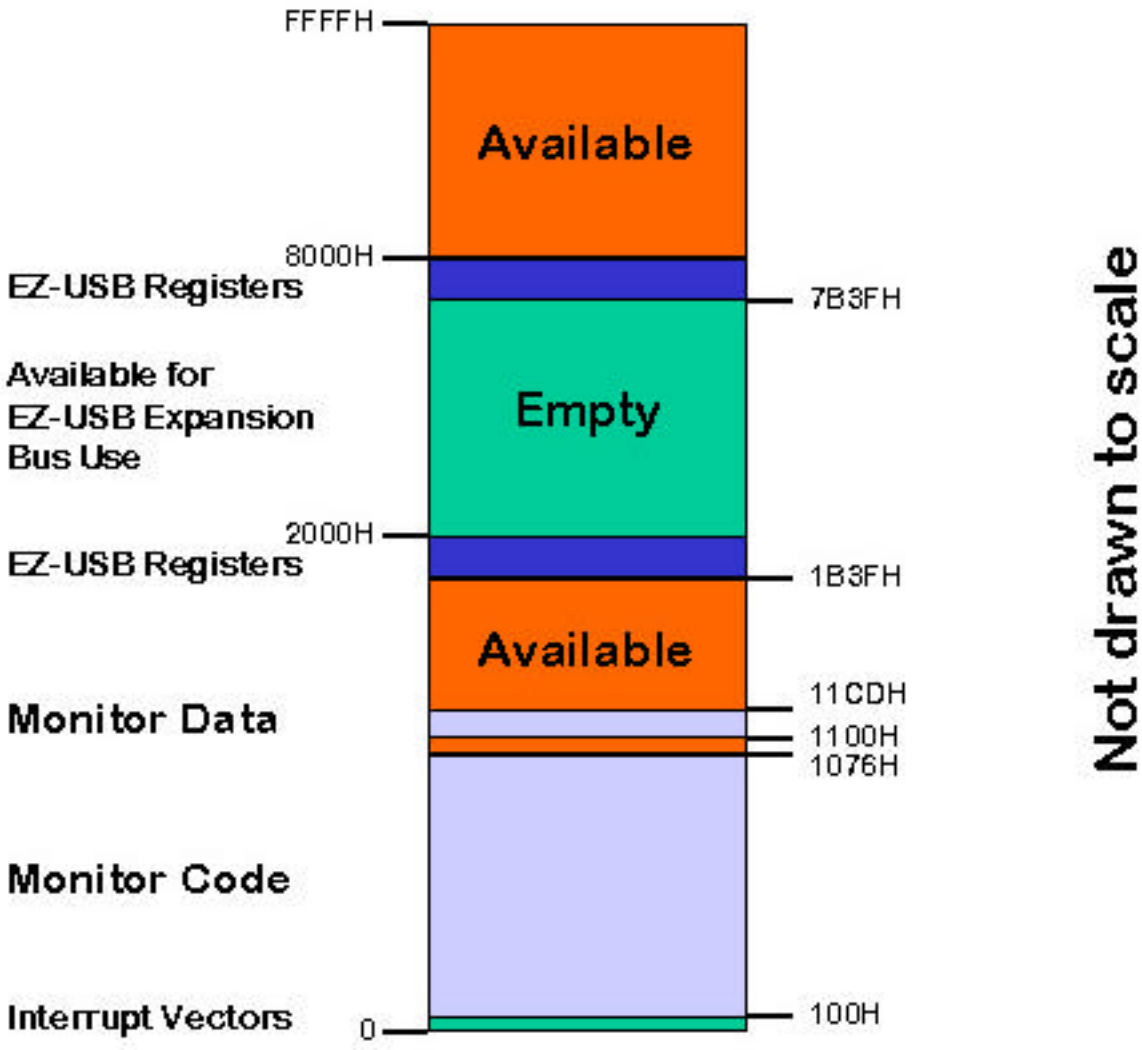


Figure 7-18 Memory map of the downloaded dScope monitor

The memory map of the USBSIMM once the dScope monitor has been loaded is shown in Figure 7-18. Note the available space in low memory at 1200H and the 32K of available memory at 8000H for a user program. Note also the "empty" region where no program memory is installed - later examples will connect external hardware to the EZ-USB expansion bus and these will be addressible in this region.

Load BAL.HEX onto the USBSIMM board but don't click GO yet.

Study the BAL.LST file that was generated in Step 2 for a moment. Notice that the first thing that the firmware does is renumerate - it programmatically disconnects itself from the PC host (which will unload the EMPTY.INF drivers) and reconnects itself to the PC host using a different set of descriptors. When you click "GO" the PC host will treat this as a totally new device and will look for a VID_PID match so that it knows which drivers to load. DO NOT USE the default Cypress VID and PID since this will create an infinite loop with LoadEZ.sys continually reloading the same program.

You may use my vendor ID (=4242H) for development but if you intend to ship a product outside of your lab then you should apply for your own Vendor ID.

In this first example a VID_PID match will not be found but the Windows Installation Wizard will continue to search looking for a suitable device driver (all attached devices MUST have a device driver). In this example, Windows will use a generic HID driver since BAL belongs to the HID class. Windows will store the fact that the BAL device is attached in an internal Plug-And-Play table. You can check that it is there by using the "System" Control Panel applet to display the current hardware configuration (Figure 7-19).

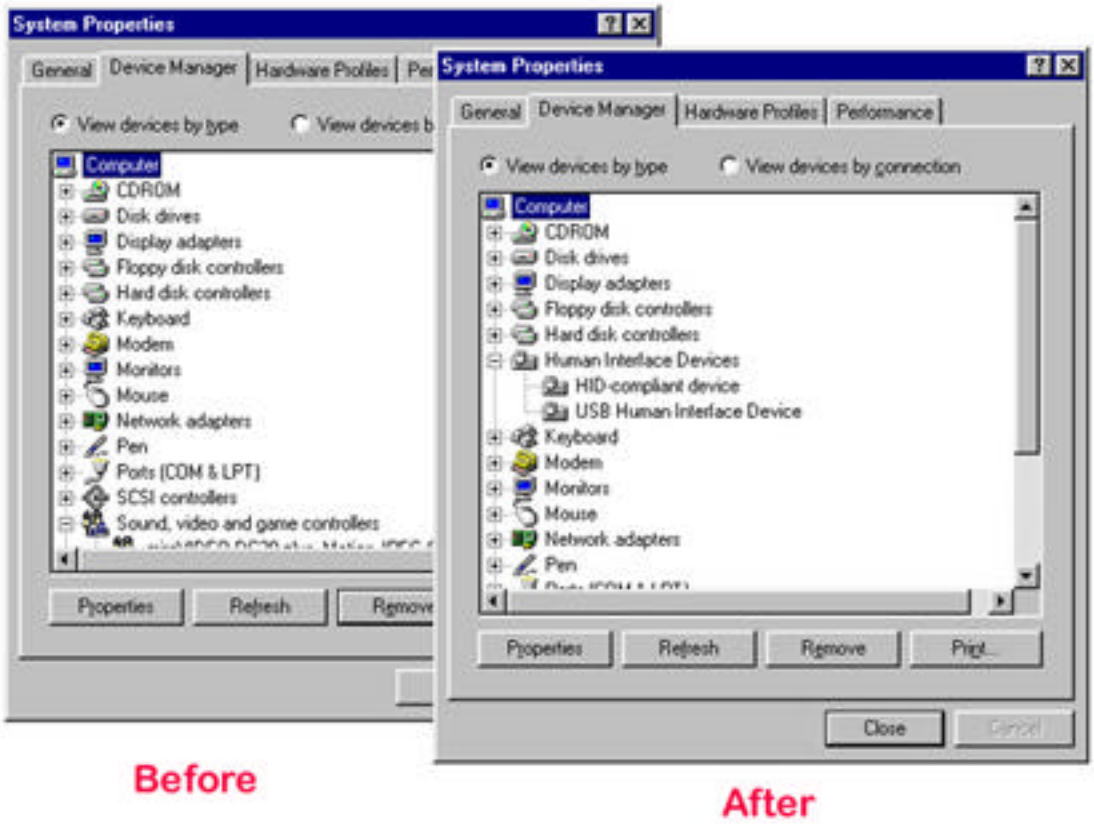


Figure 7-19 "Buttons and Lights" added to the Human Interface Devices List

The I/O device is now operational.

Step 6: Writing the PC host application

Once of the main reasons for choosing a HID implementation was the simplicity of the PC host software. Windows contains all of the device driver code to converse with HID devices and an application program can treat these devices like a "file". A connection to our specific device must first be made and then it is just a matter of ReadFile and WriteFile to send and receive reports from our IO device.

This first example uses a Visual Basic front end. The routine OpenUSBdevice

needs to search through the list of currently attached devices looking for our specific device. I chose to look for a match of the Product Description String - I felt that this was more interesting than looking for a VID, PID match. Once our IO device is successfully opened the operating system will provide us a HANDLE that is used in ReadUSBdevice and WriteUSBdevice.

The read and write requests should specify the same report length as defined in the Report Descriptor. This "buttons and lights" example uses a report length of 1 for both directions - remember to increase the length in the report descriptor and the read/write routines if your application needs to transfer more data.

The benefit of writing the PC host application in Visual Basic is that it is very easy to create an intuitive display as shown below. Another benefit is the ability to single step through the program and make changes as you move through the debug process.

The source for the Buttons and Lights application is included on the CDROM. The example can be successfully run using the "learning" version of Visual Basic

Figure 7-20 shows a screen shot of a Visual Basic application program human interface. There are two sets of buttons—one is “soft” and the other is a copy of the real buttons. The “soft” LEDs are a copy of the real LEDs and respond the same way. The LEDs are operated by the two sets of buttons (A = soft buttons, B = real buttons) as:

A only
A OR B
A AND B
A XOR B
B only

This scheme allows software to control the LEDs, hardware (the real buttons) to control the LEDs, or a combination of both. The full source code is provided on the CD-ROM.

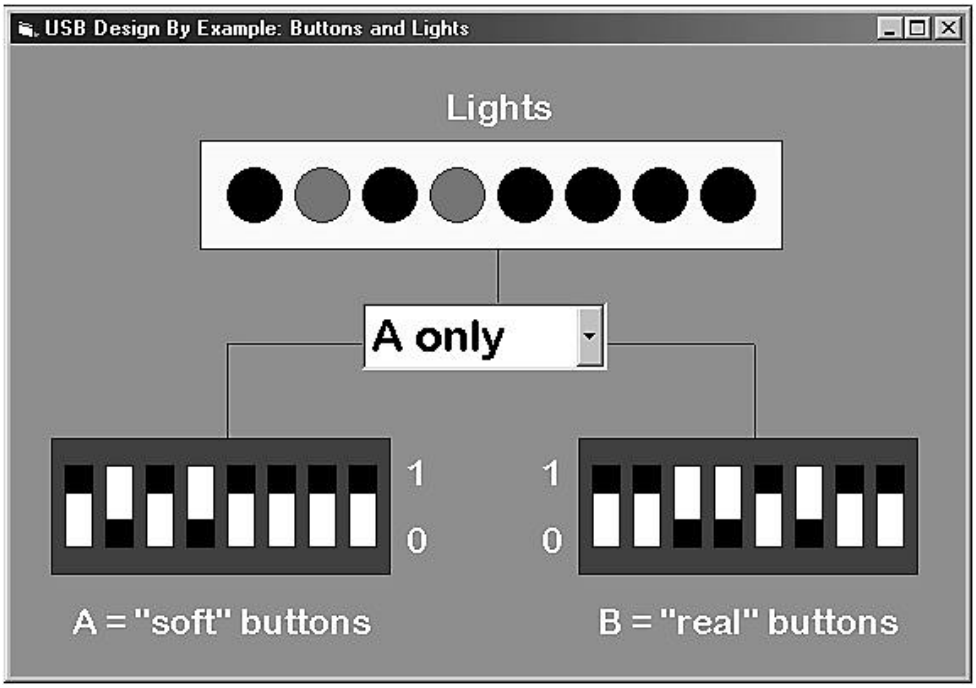


Figure 7-20. Buttons and lights application program

Design Summary

The example was a simple 8-bit input port and an 8-bit output port so we could focus on the method. We have successfully used external hardware to change the operation of PC host software. We have also used software to change some hardware external to the PC. This is a significant milestone—we have PC host software interacting with a custom I/O board. Now that we have achieved this first step, we can investigate monitoring and controlling many other external hardware examples.

CHAPTER SUMMARY

We accomplished a great deal in this chapter. We implemented the concepts we learned from previous chapters into a working "buttons and lights" example. This simple example demonstrated the PC host manipulating a physical external device (the LEDs) and demonstrated a physical external device (the BUTTONS) controlling the operation of the PC host. This milestone was made possible by the underlying infrastructure of USB.

We will build on this knowledge in the upcoming chapters. If we can operate some buttons and lights, we ought to be able to operate *anything*. . . .